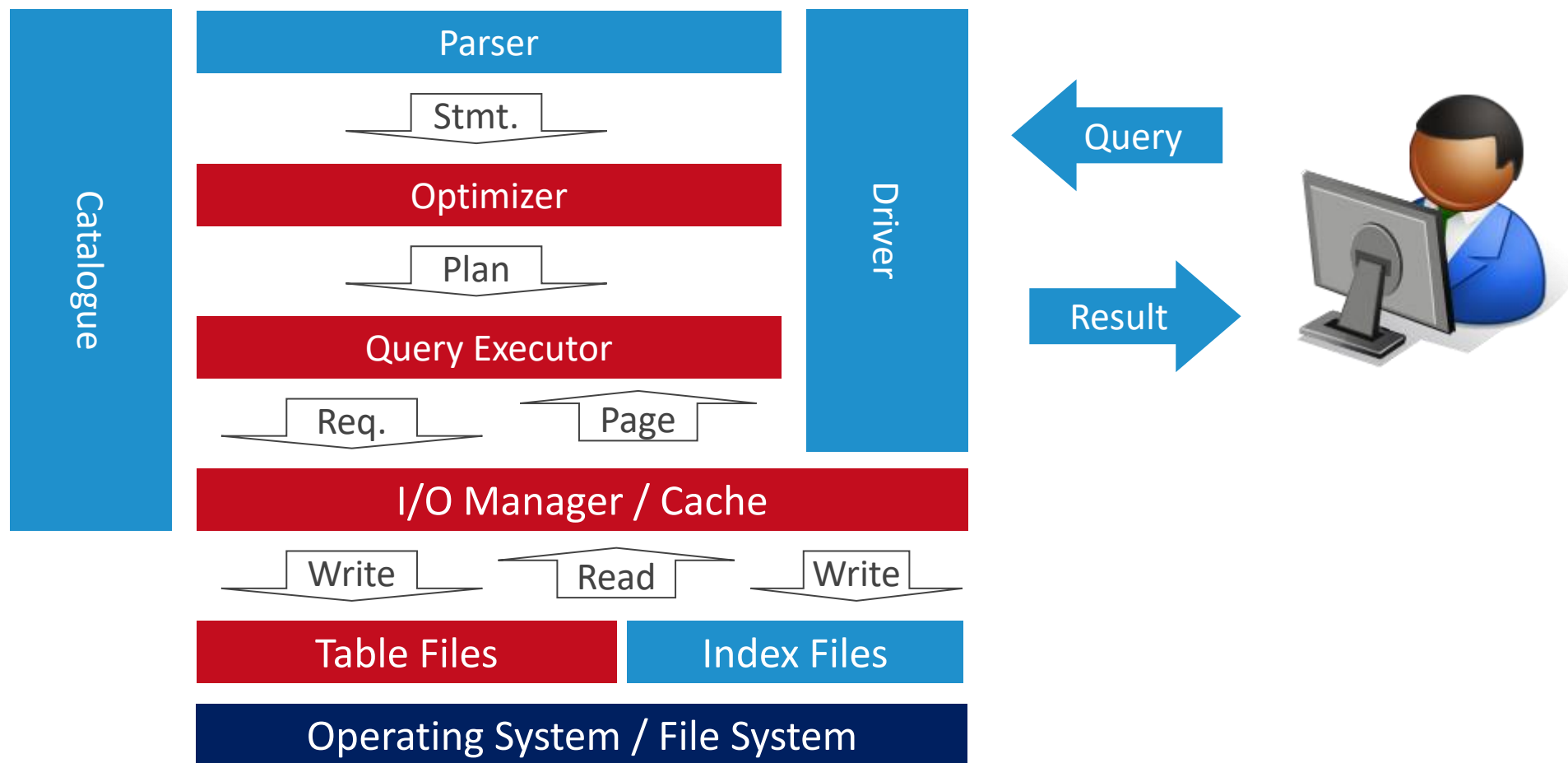


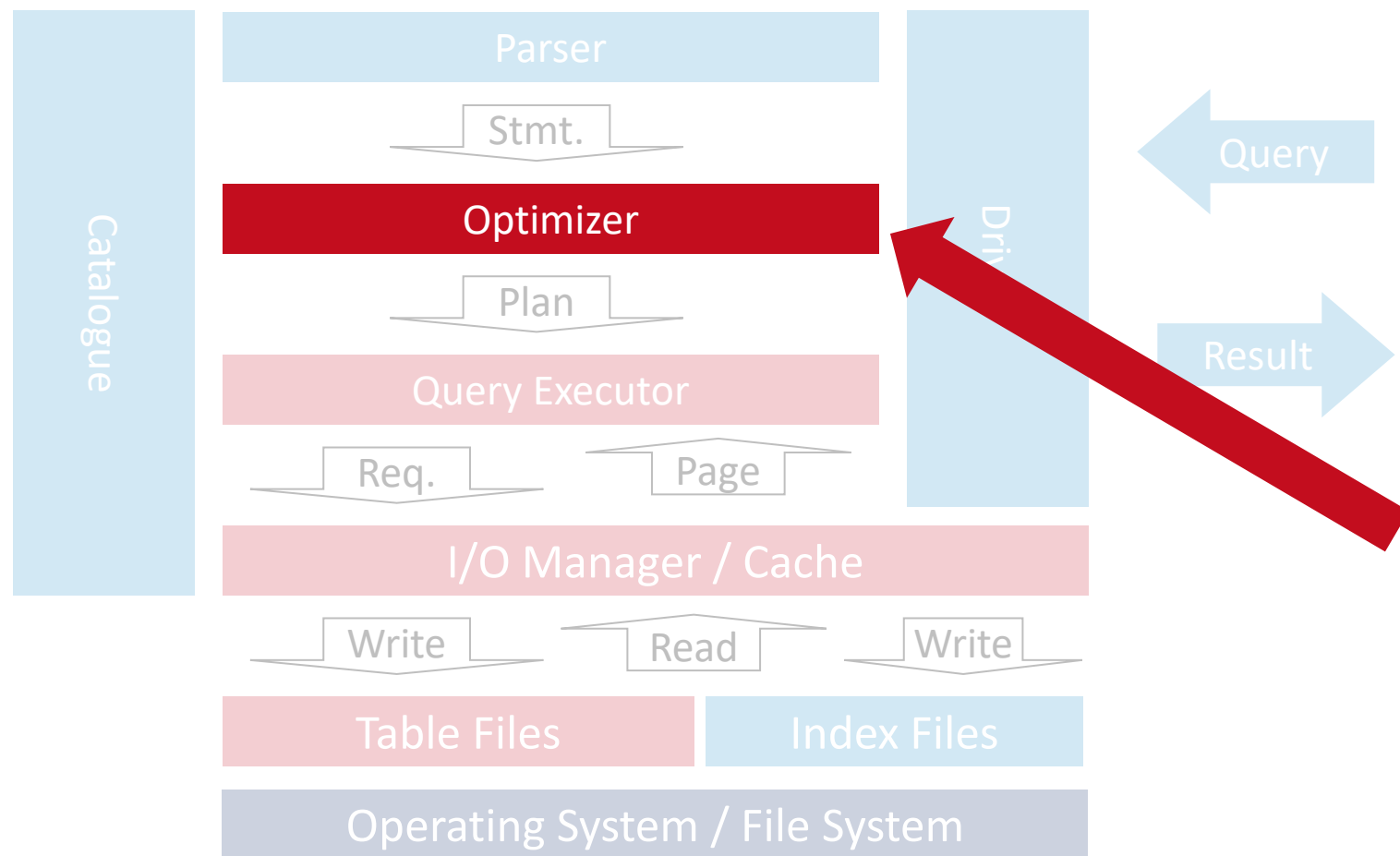
DBTLAB – Exercise 8: Query Optimization II – Cost Estimation

Rudi Poepel Lemaitre
based on material from Volker Markl

Basic system architecture



Where are we today?



Select an “optimal” physical plan
(avoid bad cases)
→ **“Reasonably-Good-imizer”**

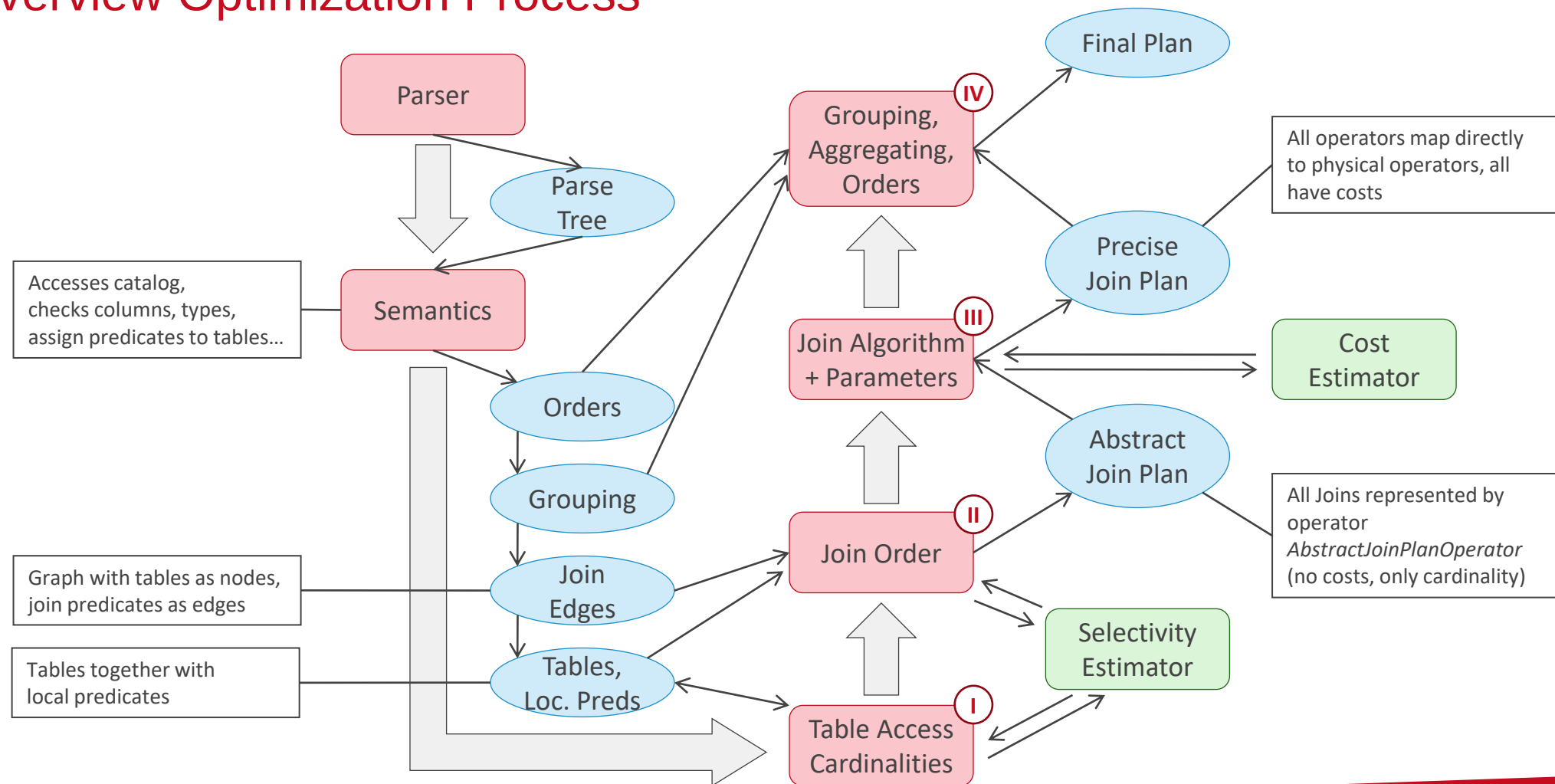
Today: Operators’ cost estimation

Query Optimization

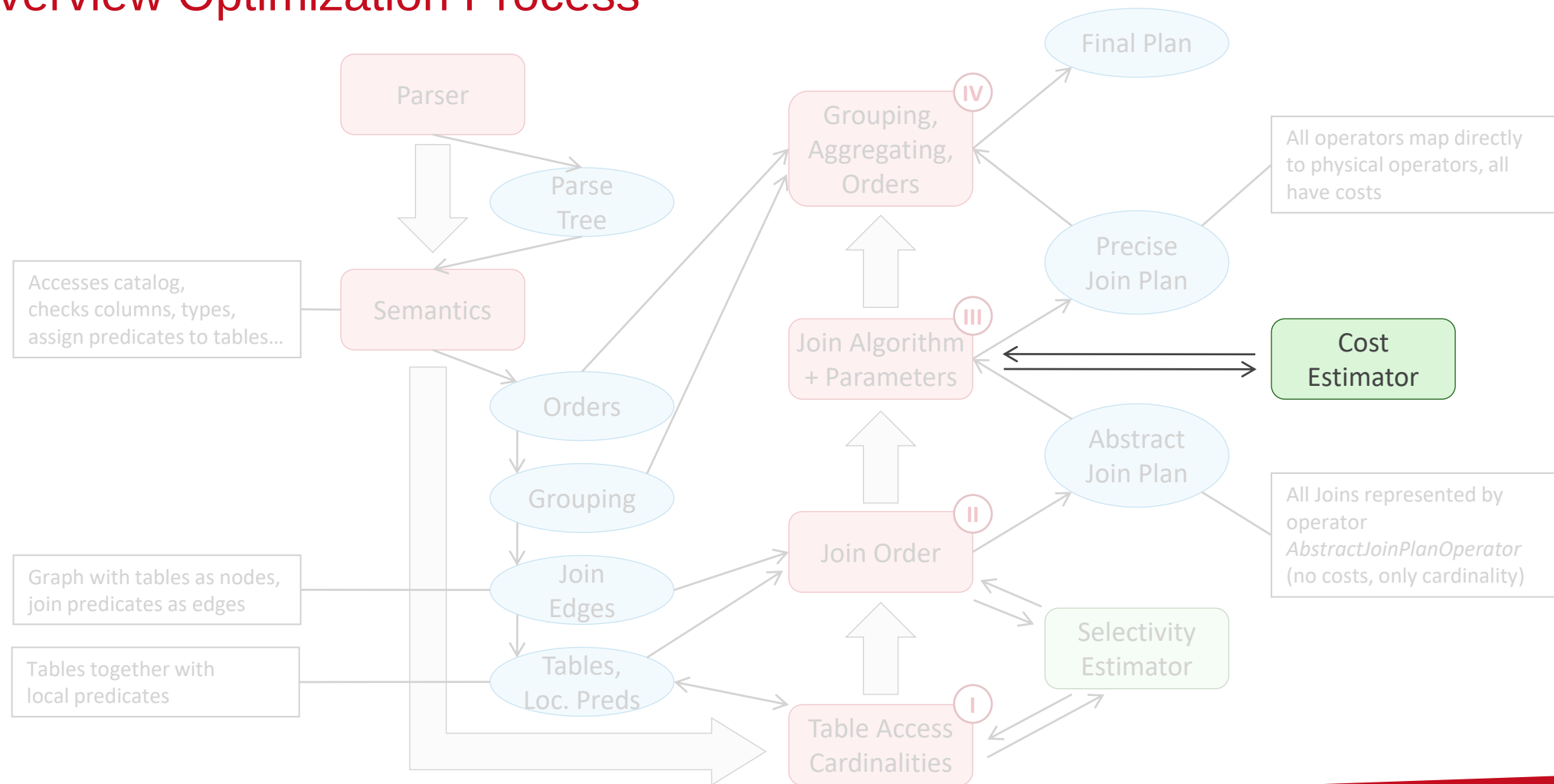
4 Steps Approach

- I) Estimation of local cardinalities
 - Cardinalities after single table access and its local predicates
- II) Cardinality-based selection of Join Order
 - Not optimal concerning total costs, but typically quite robust
 - Dynamic Programming: Optimal concerning cost formula (minimizing intermediate results)
- III) Selection of Table Access and Selection of Physical Join Operator
 - Based on I/O cost estimations
- IV) Generation of GroupBy and OrderBy
 - Just added on top of query, no analysis for push-down (sometimes aggregations are possible before the joins)

Overview Optimization Process



Overview Optimization Process



Cost Estimation

Cost based Operator Selection

- Every operator has two cost parameters
 - Operator-Costs: Costs of this single operator.
 - Cumulative-Costs: Costs of the entire sub-plan rooted at that operator.
 - When plans are compared, cumulative cost is the important measure.

Casting in Java...

- For this task, be especially careful with data types and casting.
- Many methods can become useless if do not cast properly!
- Consider the following example:

```
long cardinality = 1000;  
long fullCardinality = 10000;  
  
float ratio = cardinality / fullCardinality;
```



float ratio = ???

Casting in Java...

- For this task, be especially careful with data types and casting.
- Many methods can become useless if do not cast properly!
- Consider the following example:

```
long cardinality = 1000;  
long fullCardinality = 10000;  
  
float ratio = cardinality / fullCardinality;
```



float ratio = 0.0 !!!!!!!!!!!

```
long cardinality = 1000;  
long fullCardinality = 10000;  
  
float ratio = (float)cardinality / (float)fullCardinality;
```



float ratio = 0.1

Operator-Cost: Table Scan

- We assume a simple sequential scan.
- The overhead of the random access between the sequential sub-sequences is assumed to be hidden through prefetching.
- However, the cost of adjusting the disk head before reading the first block should be considered.
- Formula:

$$C_{TableScan} = C_{rand_r} + \frac{B_{PS_table}}{B_{PS_default}} * C_{read} * N_{pages}$$

No casting
required

C_{rand_r} = random read overhead

B_{PS_table} = number of bytes from the table's page size

$B_{PS_default}$ = number of bytes from the default page size

C_{read} = cost of reading a single block of the default size

N_{pages} = number of pages to read

Operator-Cost: Index Lookup

- We assume that the costs of an index scan are to descend the B-Tree (random I/O, minus pages from the first two levels, which are in the cache) and then the scan along the leaves,
- Formula:

$$C_{idx_lookup} = C_{rand_r} + \left((N_{levels} - 2) * (C_{rand_r} + C_{read_page}) + C_{rand_r} \right) + C_{read} * N_{leaves}$$

$$C_{read_page} = \frac{B_{PS_idx}}{B_{PS_default}} * C_{read}$$

$$N_{leaves} = \left\lceil \frac{Card_{est}}{Card_{full}} * N_{total_leaves} \right\rceil$$

Casting
required

N_{levels} = total number of levels in the B-Tree

C_{read_page} = Cost of reading a page from the index

N_{leaves} = number leaves to read

$Card_{est}$ = Estimated output cardinality (given as parameter)

$Card_{full}$ = Total cardinality of the base table

N_{leaves} = total number of leaves in the B-Tree

Operator-Cost: Index Lookup

Casting
required

$$C_{idx_lookup} = C_{rand_r} + \left((N_{levels} - 2) * (C_{rand_r} + C_{read_page}) + C_{rand_r} \right) + C_{read} * N_{leaves}$$



Cast result to `long`. No intermediate casts are required.

Operator-Cost: Index Lookup

Casting
required

$$C_{idx_lookup} = C_{rand_r} + \left((N_{levels} - 2) * (C_{rand_r} + C_{read_page}) + C_{rand_r} \right) + C_{read} * N_{leaves}$$



If $N_{levels} - 2 < 0$, then all this expression is equals to 0.

Operator-Cost: Index Lookup

Casting
required

$$N_{leaves} = \left[\frac{Card_{est}}{Card_{full}} * N_{total_leaves} \right]$$



Cast result to `float`. Also, cast internally every expression to `float`.

Operator-Cost: Index Lookup

Casting
required

$$N_{leaves} = \left\lceil \frac{Card_{est}}{Card_{full}} * N_{total_leaves} \right\rceil$$



Cast result to `int`. Use `Math.ceil` to round up.

Operator-Cost: Sort

- For simplicity and robustness, the sort is assumed to be always external and two-phase multi-way.
- All tuples are written in blocks once in a sequence and then read again.
- We assume that all writes occur directly one after the other, so only one random write overhead should be considered.
- For the read phase, all blocks are read once. The sort code can utilize some form of prefetching on the blocks, but random access is not completely excluded.
- We assume that each block read has an associated random read overhead, accounting for one-sixteenth of the random read overhead for the default page size.

Operator-Cost: Sort

– Formula:

Casting
required

$$C_{sort} = C_{writing_blocks} + C_{reading_blocks}$$

$$C_{writing_blocks} = N_{blocks} * \frac{B_{PS_heap}}{B_{PS_default}} * C_{write} + C_{rand_w}$$

$$N_{tuples/block} = \frac{B_{PS_heap} - B_{header}}{B_{tuple}}$$

$$C_{reading_blocks} = N_{blocks} * \left(\frac{B_{PS_heap}}{B_{PS_default}} * C_{read} + \frac{C_{rand_r}}{16} \right)$$

$$N_{blocks} = \frac{N_{tuples}}{N_{tuples/block}}$$

B_{PS_heap} = number of bytes from the heap's page size

C_{write} = cost of writing a single block of the default size

C_{rand_w} = random write overhead

B_{header} = header size in bytes (use `TablePage.TABLE_DATA_PAGE_HEADER_BYTES`)

B_{tuple} = bytes required to store a single tuple (use the `QueryHeap.getTupleBytes()` method)

Operator-Cost: Sort

– Formula:

Casting
required

$$N_{tuples/block} = \frac{B_{PS_heap} - B_{header}}{B_{tuple}}$$



Don't cast anything! $N_{tuples/block}$ can be either a **long** or **int**, so
leave every expression as a **long** or **int**.

The integer division will take care of rounding down $N_{tuples/block}$.

Operator-Cost: Sort

– Formula:

$$N_{blocks} = \frac{N_{tuples}}{N_{tuples/block}}$$



If $N_{blocks} = 0$, then use $N_{blocks} = 1$.

Casting
required

Operator-Cost: Fetch

- If the Fetch operator receives a sorted input after the RID, it uses a model where no page is assumed to be read twice.
- The cost of seeking a page goes down with the density of the fetches due to the file system and disk cache evenly prefetching.
- The cost model is the following:
 - Let us assume we have N_{pages} pages in total and a cardinality of $Card$.
 - For the first tuple, we must fetch a page.
 - For the next tuples, we only need to fetch a new page with a certain probability, which depends on the number of pages we have seen already.
 - This leads to a recursive formula for the number of accessed pages (i being the current number of processed tuples):

$$N_{read_pages}(i + 1) = N_{read_pages}(i) + \left(1 - \left(\frac{N_{read_pages}(i)}{N_{pages}} \right) \right)$$

Operator-Cost: Fetch

- Solving the recursion gives us:

$$N_{read_pages}(i) = \left(\frac{1}{k} - \frac{1}{k - k^2} \right) * k^i + \frac{1}{1 - k} \quad k = \frac{N_{pages} - 1}{N_{pages}}$$

Casting
required

- Further, the ratio of the pages that are accessed randomly by the fetch operator is approximated by the following formula:

$$R = 1 - \left(\frac{N_{read_pages}}{N_{pages}} * \frac{7}{8} \right)$$

- Total cost when the input is sorted after the RIDs:

$$C_{fetch} = N_{read_pages}(Card) * (C_{read} + C_{rand_r} * R)$$

Operator-Cost: Fetch

Casting
required

$$k = \frac{N_{pages} - 1}{N_{pages}}$$



k 's datatype is `double`. Cast internally every expression to `double`.

Operator-Cost: Fetch

Casting
required

$$N_{read_pages}(n) = \left(\frac{1}{k} - \frac{1}{k - k^2} \right) * k^n + \frac{1}{1 - k}$$



Cast result to `double`. No intermediate casts are required.

Operator-Cost: Fetch

Casting
required

$$N_{read_pages}(n) = \left(\frac{1}{k} - \frac{1}{k - k^2} \right) * k^n + \frac{1}{1 - k}$$



Cast result to `double`. No intermediate casts are required.

Operator-Cost: Fetch

Casting
required

$$N_{read_pages}(n) = \left(\frac{1}{k} - \frac{1}{k - k^2} \right) * k^n + \frac{1}{1 - k}$$



Cast result to `double`. Use `Math.pow`.

Operator-Cost: Fetch

Casting
required

$$N_{read_pages}(n) = \left(\frac{1}{k} - \frac{1}{k - k^2} \right) * k^n + \frac{1}{1 - k}$$



Cast result to `long`.

If $N_{read_pages}(Card) < 1$, use $N_{read_pages}(Card) = 1$ instead.

If $N_{read_pages}(Card) > N_{pages}$, use $N_{read_pages}(Card) = N_{pages}$ instead.

Operator-Cost: Fetch

Casting
required

$$R = 1 - \left(\frac{N_{read_pages}}{N_{pages}} * \frac{7}{8} \right)$$



R's datatype is `double`. Cast internally every expression to `double`.
INCLUDING THE NUMBERS!

Operator-Cost: Fetch

Casting
required

$$C_{fetch} = N_{read_pages}(Card) * (C_{read} + C_{rand_r} * R)$$



Cast result to long.

Operator-Cost: Fetch

- If the FETCH receives RIDs that are not sorted, then it must access each page individually.
- Here, the cache hit ratio increases when the number of fetched tuples increases.
- Formula:

$$C_{fetch_unsorted} = Card * (1 - R_{hit}) * (C_{rand_r} + C_{read})$$

$$R_{hit} = \frac{\ln(Card)}{\ln\left((10 * N_{pages})^{\frac{1}{0.66}}\right)}$$

R_{hit} = cache hit ratio

Casting
required

Operator-Cost: Fetch

Casting
required

$$R_{hit} = \frac{\ln(Card)}{\ln\left((10 * N_{pages})^{\frac{1}{0.66}}\right)}$$



R_{hit} 's datatype is **double**. Cast internally every expression to **double**.

INCLUDING THE NUMBERS!

If $R_{hit} > 0.8$, use $R_{hit} = 0.8$ instead.

Operator-Cost: Fetch

Casting
required

$$C_{fetch_unsorted} = Card * (1 - R_{hit}) * (C_{rand_r} + C_{read})$$



Cast result to `long`.

Operator-Cost: Filter, MergeJoin, and NestedLoopJoin

- We are only considering I/O costs. Therefore, we assume that a Filter has a cost of 0.
- The MergeJoin is also assumed to be free because it has no I/O costs. All cost is represented in the required sort operators.
- The cost from a NestedLoopJoin is directly derived from the operator semantics and assumes that the inner child is executed once for each tuple of the outer side.
- NestedLoopJoin formula:

$$C_{NL_join} = Card_{outer} * C_{cumulative_inner}$$

$Card_{outer}$ = cardinality of the outer side.

$C_{cumulative_inner}$ = cumulative cost from the inner side.

No casting
required

Exercise 8

Exercise 8

- Implement a cost estimator using the following interface:
 - `de.tuberlin.dima.minidb.optimizer.cost.CostEstimator`
- Implement the following factory method:
 - `createCostEstimator`

Tests and points

Test name	Points	Description
testComputeTableScanCosts	0,5	Tests the table scan's cost of multiple tables.
testComputeIndexLookupCosts	1,5	Tests the index scan's cost of multiple indexes.
testComputeSortCosts	2	Tests the sorting cost of multiple tables and attributes.
testComputeFetchCosts	2,5	Tests the fetch cost of multiple configurations.
testComputeFilterCost	0	Tests the filter cost.
testComputeMergeJoinCost	0	Tests the cost of a MergeJoin.
testComputeNestedLoopJoinCost	0,5	Tests the joining cost of multiple tables and attributes.